DIGITAL ACCESS TO SCHOLARSHIP AT HARVARD
DASH.HARVARD.EDU

HARVARD LIBRARY
Office for Scholarly Communication

Two Applications of Hand-Printed Two-Dimensional Computer Input

Citation
Lewis, Harry R. 1968. Two Applications of Hand-Printed Two-Dimensional Computer Input. Bachelor's thesis, Harvard College.

Permanent link
http://nrs.harvard.edu/urn-3:HUL.InstRepos:12220368

Terms of Use
This article was downloaded from Harvard University's DASH repository, and is made available under the terms and conditions applicable to Other Posted Material, as set forth at http://nrs.harvard.edu/urn-3:HUL.InstRepos:dash.current.terms-of-use#LAA

Share Your Story

The Harvard community has made this article openly available.
Please share how this access benefits you. Submit a story .

Accessibility

Two Applications of
Hand-Printed Two-Dimensional Computer Input

A thesis presented

by

Harry Roy Lewis

to

The Committee on Applied Mathematics

in partial fulfillment of the honors requirements

for the degree of

Bachelor of Arts

Harvard College

Cambridge, Massachusetts

May 1, 1968

Acknowledgments

The debts which I owe to Mr. Robert Anderson and Mr. Kenneth Ledeen are even greater than is obvious from the citations in this thesis. As well as providing me with excellent subroutines, flowcharts, and algorithms which were incorporated into these programs, they gave me a great deal of advice about general programming techniques for two-dimensional input and output, and their enthusiasm is largely responsible for my continued interest in this project.

So much of whatever I know about computing in general and graphics in particular is due to the teaching and advice of Prof. Ivan Sutherland that it would be impossible to say which specific ideas described here are really his, rather than mine. In a word, his contributions have been numerous. I would also like to thank Prof. Anthony Oettinger and Mr. Thomas Cheatham, Jr. for their encouragement, suggestions, and general assistance.

The classroom use of SHAPESHIFTER was in Applied Mathematics 201, taught at Harvard University by Prof. George Carrier; I thank him for the opportunity to experiment with that program on a "live" class.

This project has been supported in part by the Advanced Research Projects Agency under Contract SD-265.

Table of Contents

Illustrations

Introduction

According to a traditional definition, the object of technology is to extend the capacities of man's senses and the power of his muscles. The computer is commonly regarded, by analogy, as a prosthesis for the brain. Yet man's anatomy greatly influences his problem-solving process; his body acts as more than a depository and processing machine for abstract information. His nervous system is not merely an isolated mind, but is a functional unity innervating sensors and muscles. His eyes and hands are not only the organs by means of which symbols are interpreted and manipulated, but also the equipment which molds the ways in which thoughts are thought and concepts are conceptualized. These are basic and inescapable, though poorly understood, facts of human psychology.

To assist human thought by machine, then, means more than to perform logical manipulations too complicated or numerous for man's brain. In addition the machine must be made to co-operate with all the aspects of the human thought process as they actually function. The problems described in this thesis are therefore part of the larger challenge to the future designers of computer systems: to make the computer a partner in the human thought process by modelling its forms of representation after those which, either because of cultural tradition or innate psychology, are used by man.

In the past the computer has been of little help in the solution of poorly formulated problems. When working on such a problem the

researcher usually does not know precisely what questions he wants to ask. He could best be assisted by a system in which he can experiment and pose a variety of questions. He should get answers quickly so that he can change his questions according to the computer's responses. Together, then, the man and machine could close in on a solution.

The languages conventionally used for describing questions to a machine bear little similarity to those designed by people for communication with each other. Rapid man-machine interaction requires that the machine be able to accept and describe information in a form more easily understood by the man. The "natural" language for man is often pictorial or two-dimensional, while the machine is usually constrained to a one-dimensional format. Algebraic notation is a good example; the fact stated by a mathematician as

$$y = \sum_{n=1}^{5} x^n$$

must be rewritten for a computer as

$$y = 0 \ ; \text{ for } n = 1 \text{ step } 1 \text{ until } 5 \text{ do } y = y + x^n$$

If only the computer could understand graphical input and respond in kind, it would be a far more useful tool for interactive problem-solving.

The hardware needed for more convenient systems is now available. Computers are accessible to people in almost every field of scientific research. Cathode-ray tubes make possible the representation of data in pictorial form. Tablet devices, such as the RAND tablet used in the programs discussed here, furnish two-dimensional input, just as the cathode-ray tube provides graphical

output. A user can write on the tablet surface with a special stylus as with an ordinary ball-point pen.

The problems of utilizing a two-dimensional input medium fall into two categories:

(1) Recognition of individual hand-printed characters. This is a problem of data analysis; the geometry of a set of points must be matched to an entry in a dictionary.

(2) Recognition of two-dimensional character configurations. This is a new form of syntactic analysis; for example, rules and techniques must be specified for the parsing of mathematical expressions.

It may be that this division does not capture the essence of the problems; there may be intermediate cases, or one category may be reducible to the other, even though they seem to be so different. Useful techniques, however, have been developed both for character recognition [6] and for the parsing of two-dimensional mathematics [2], and the nature of these methods support the distinction. The research explained below deals mainly with problems of the second category.

This thesis describes two programs which use two-dimensional input and output facilities and enable a person relatively unfamiliar with computers to pose complex problems. Part I describes an application to mathematics education; Part II, the design of a two-dimensional programming language.

Part I. SHAPESHIFTER

§1. Purpose

SHAPESHIFTER shows graphically the effects of various transformations applied to a set of points in the complex plane. The user prints in his own handwriting a formula or series of formulas which serves as the real-time input to the program. By experimentally changing the transformation he can gain an understanding, difficult to achieve formally, of its general properties, the effects of parameters and signs, the location and nature of singular points, conformality, etc.

Consider, for example, an engineer trying to design an airplane wing. Carrier [3, p. 157] suggests how conformal-mapping techniques could be used to solve this problem. The engineer would have general ideas about the desired shape and want to find a mathematical formula which would generate that curve from a circle. Knowing the transformation, he could then reverse the process to compute the aerodynamic characteristics of the airfoil.

SHAPESHIFTER would be helpful to the engineer working on this problem for several reasons:

- The arduous task of computing values of functions is done automatically.
- The results are presented exactly as desired -- as a picture of a cross-section of the wing. (Figures 1, 2, and 3 illustrate what he might see: figure 1 shows a domain in the complex plane, figure 2 a hand-printed formula, and

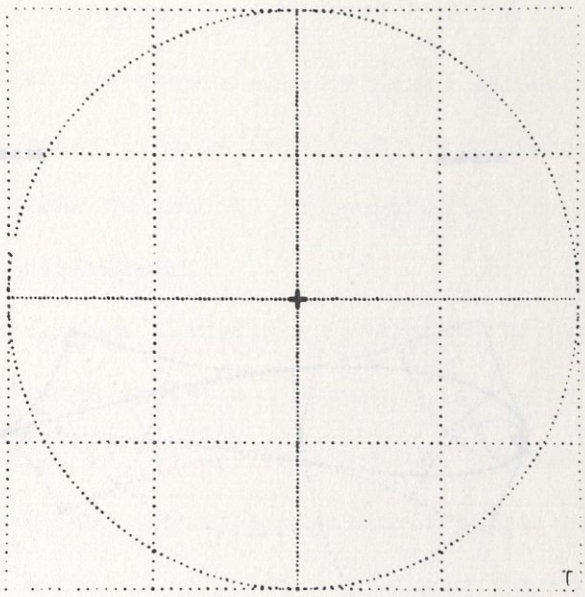
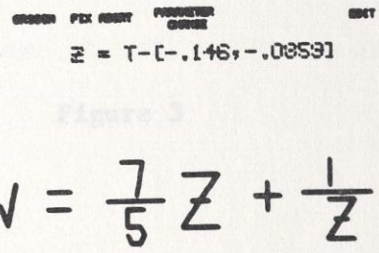


Figure 1



$$W = \frac{7}{5}Z + \frac{1}{Z}$$

Figure 2

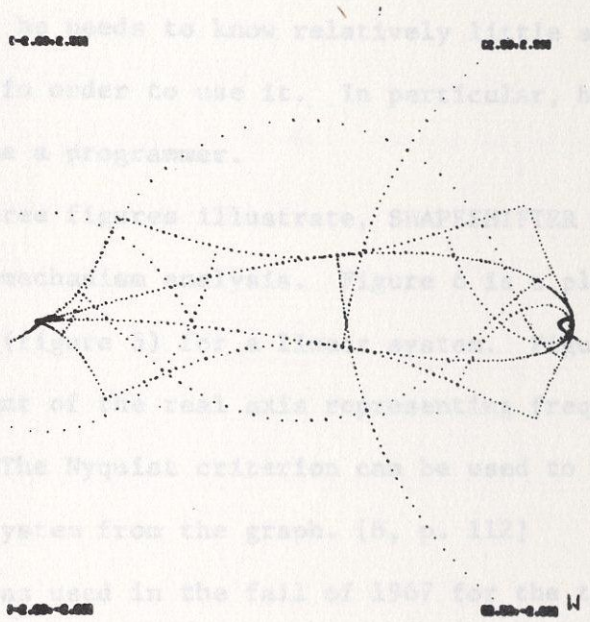


Figure 3

figure 3 an airfoil which is the range of this function restricted to the given domain.)

- Since he can write the equation in ordinary mathematical notation, he needs to know relatively little about the computer in order to use it. In particular, he does not need to be a programmer.

As the next three figures illustrate, SHAPESHIFTER can also be applied to servomechanism analysis. Figure 6 is a plot of the open-loop equation (figure 5) for a linear system. Figure 4 shows the domain, a segment of the real axis representing frequencies in the interval (-2, 2). The Nyquist criterion can be used to determine the stability of this system from the graph. [8, p. 112]

SHAPESHIFTER was used in the fall of 1967 for the teaching of conformal mapping in a course on the applications of complex-variable theory to physics and engineering. Although the educational effectiveness of this classroom session is not known, the program has taught the author (a veteran of this course) a great deal about conformal mapping.

§2. Using SHAPESHIFTER

The user is seated in front of the three scopes and a RAND tablet, on which he can write with the special stylus. One of the scopes displays a portion of the complex domain to be processed; another displays a portion of the resulting complex range; and the third scope displays a "control panel." On the control panel formulas appear as they are written and several messages are shown.

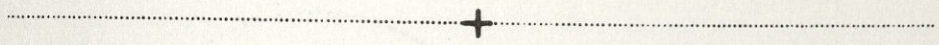


Figure 4

$$W = \frac{-Z^2}{(1+2iZ)(1-\frac{iZ}{2}-Z^2)}$$

Figure 5

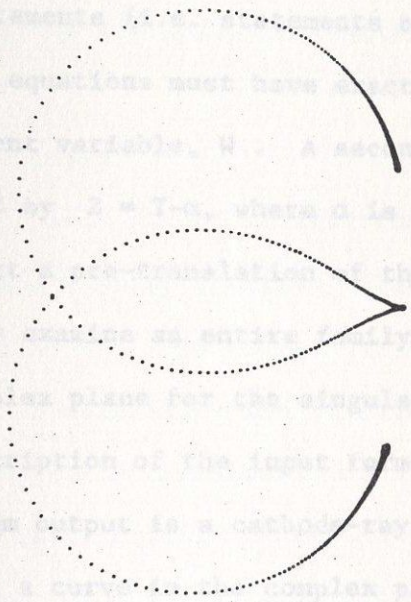


Figure 6

These messages act as buttons by which the user directs the operation of the program; a cross representing the position of the Grafacon stylus is moved until it is over the desired message, the stylus is pressed down, and the program takes the indicated action.

A transformation is described to the computer as a system of assignment statements (i.e. statements of the form "<variable>=<expression>"). Together these equations must have exactly one independent variable, Z, and one dependent variable, W . A secondary independent variable, T, is related to Z by $Z = T-\alpha$, where α is a parameter which can be varied to effect a pre-translation of the domain. By this means the user can easily examine an entire family of transformations, or he can search the complex plane for the singular points of any particular one. A complete description of the input format is in Appendices 2 and 6.

The program output is a cathode-ray tube display of a set of points defining a curve in the complex plane. The T and W planes are viewed through magnifying windows of adjustable size and position. Also displayed is a FORTRAN-type linear string version of the equations.

SHAPESHIFTER enters three principal phases of operation in cyclic order. In the ground state the user can perform initializations and change the size and position of the window through which he views the domain. In the character entry phase he prints the formulas which specify the transformation. In the execution phase, the value of the transformation is computed for each point in the domain and this result is plotted on the range scope. From the execution state the user may return to the entry phase to edit the

formulas he has written, or he may re-enter the ground state by either accepting or rejecting the output curve as the domain for the next transformation. Functions may thus be composed during successive cycles of operation.

A complete description of the program's operation is in Appendix 1. Appendix 7 gives the operating instructions for the program.

§3. Internal Structure

SHAPESHIFTER consists of several independent modules: control, computational, and display routines, character recognizer, parser, and independent display and data files. Of these parts only two are implementations of relatively new techniques: the routine which recognizes hand-printed characters and the routine which parses two-dimensional mathematical expressions. The character recognizer is a working program supplied by Kenneth Ledeen [6]. The major effort of writing SHAPESHIFTER was the implementation of the parsing algorithm invented by Robert Anderson [2]. The parsing algorithm is described in §6.

Data in SHAPESHIFTER are represented in two forms: floating-point and fixed-point. The main storage of a point is as a pair of floating-point numbers in the data file. The points in the display file are represented in fixed-point form; these numbers are derived from the values in the data file. Either of two "standard" domains can be selected at initialization: a circle of unit radius (figure 7), or that circle surrounded by ten grid lines (figure 3).

The computational subroutines, drawn largely from algorithms

in references [1,4,5], are applied to the numbers in the data file. These subroutines are slow, since floating-point hardware is not available on this machine.

The points are stored in both the data and display files in random order.

- Processing the points from the domain in random order causes the shape of a new curve to become evident quickly, even if computation of the entire range takes a relatively long time. When computation is initiated or a parameter is changed, points disappear at random from the old curve and reappear in new positions; the previous curve seems to fade out, and the new one to grow in intensity.
- Displaying the points in random order reduces the apparent flicker of the scopes, which would otherwise be quite annoying. In each short interval of time the eye sees some points from all parts of the curve, even though it takes a perceptibly long time to display all the points on the curve. Therefore no portion of the display seems to flicker badly; rather, the entire display twinkles.

§4. Controls

Control over the program is achieved by use of the Grafacon stylus, several toggle switches, and a "joystick" which is described below. The operation of any of these devices changes the display, so that the user is given instant feedback about anything he does which affects the program.

The Grafacon stylus serves several purposes. It is used to press the buttons which monitor the overall flow of control within the program; in each state messages are displayed which name the alternative states in which the user may next place the program.

The stylus is also used for three kinds of "following" operations similar to light-pen tracking. In the first, the stylus is used simply to position a cross on the control scope in the location corresponding to that of the stylus over the tablet. (The size and co-ordinate system of the tablet are identical to those of the scope.) No trail is left by the cross as it moves. In this case hand-eye co-ordination is almost unconscious; the scope and tablet are psychologically merged and the position of the stylus is the position of the cross.

In the second case, the positions of the stylus and cross do not correspond identically; they are separated by a constant vector. When the stylus is used to change the value of a complex-valued parameter which is represented as a point on the complex plane and a cross on the scope, the virtual position of the tablet surface as projected onto the scope is determined when the stylus is first pressed down. The tablet co-ordinates are translated so that the location of the stylus at that time corresponds to the location of the cross on the scope; subsequent motions of the stylus about this point on the tablet cause corresponding motions of the cross relative to its initial position. By this technique the cross can be moved a small distance from its initial position, no matter where the stylus happens to be set down. Since the position of the cross never changes

discontinuously, the user can make fine adjustments of the value of the parameter.

The third type of "following" is in the character recognizer. As in the first type, a spot of light on the scope corresponds identically to the position of the stylus over the tablet, but when the stylus is pressed down and moved a trail of "ink" appears which represents its trajectory. Once again, hand-eye co-ordination is a minor problem when this feedback is provided.

§5. Output View Selection

The size and position of the square window through which the complex plane is viewed are determined by the position of the window's center as specified by a complex-valued parameter, and the width (or height) of the window as specified by a real-valued parameter. The first parameter is adjusted by a "joystick" -- two potentiometers mounted so that they are controlled by means of a common shaft which can move freely in a solid angle. The second parameter is changed via two toggle switches: closing the switches increases or decreases the size of the window at an exponential rate.

The psychological effect of the windowing controls is an "outside-in" view of the complex plane; that is, the plane is fixed, and the user adjusts the window to inspect various parts of it. To move the window (not the plane) in some direction, he pushes the joystick in that direction; the picture appears to move in the opposite direction. When the size-increasing switch is closed, the window's dimensions always double in equal time intervals, so that points

seem to move across the display at the same velocity, no matter how large the window is.

Five views of the result of the transformation of figure 8 applied to the domain in figure 7 are shown in figures 9 - 13. The center of the window is in each case at the origin of the complex plane; the width of the window is respectively 2000, 200, 20, 2, and .2 . These pictures point out the importance of windowing; the appearance of the curve in the large and in the small differs considerably, and none of the views shows both.

Figure 13 illustrates another interesting characteristic of the output. Since the domain is a set of uniformly spaced points, rather than a set of connected line segments, the spacing of the points in the range changes gradually under any continuous transformation. The display therefore provides additional information by suggesting how the complex plane is stretched and distorted. Consequently the output curve sometimes appears three-dimensional.

§6. Parser

The most important innovation in SHAPESHIFTER is the use of ordinary mathematical notation for writing formulas. The algorithm which parses hand-printed two-dimensional expressions is adapted from one developed by Robert Anderson [2, Appendix I]. Anderson has developed a very general parsing algorithm for two-dimensional character configurations, but due to the simplicity of the types of expressions which are used here (essentially those of high-school algebra) this generality is not necessary; therefore a more efficient

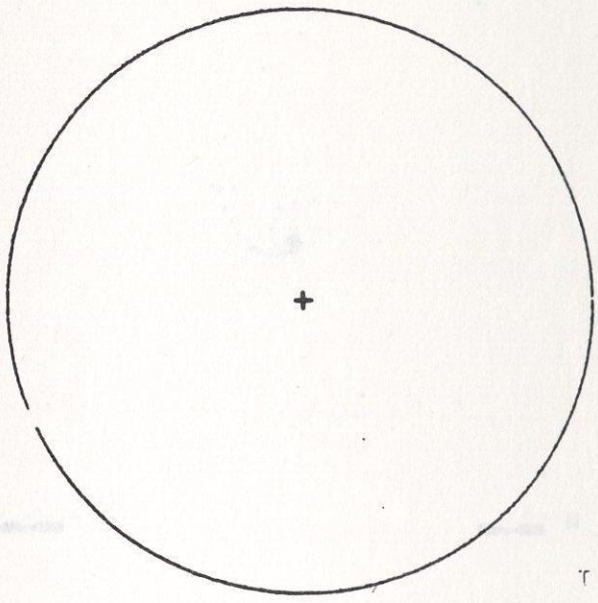


Figure 7

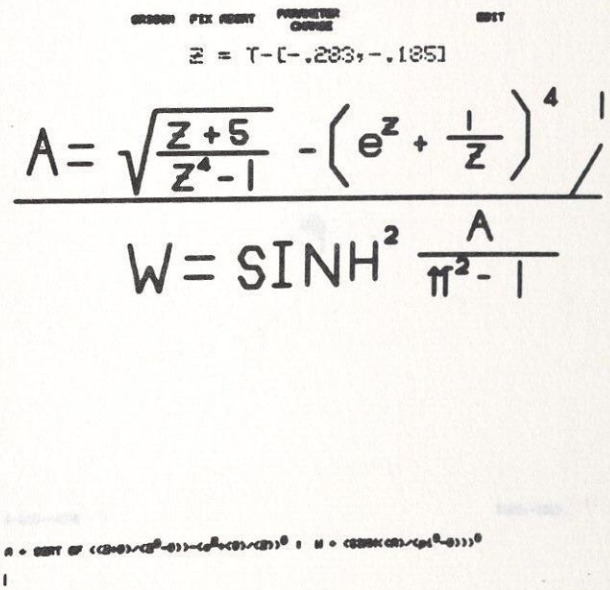


Figure 8

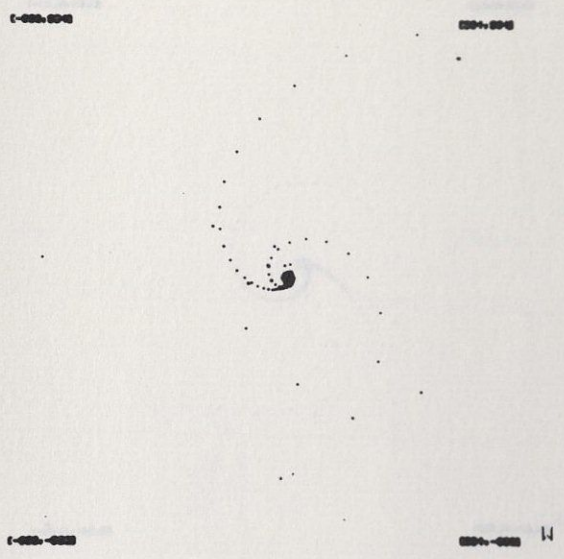


Figure 9

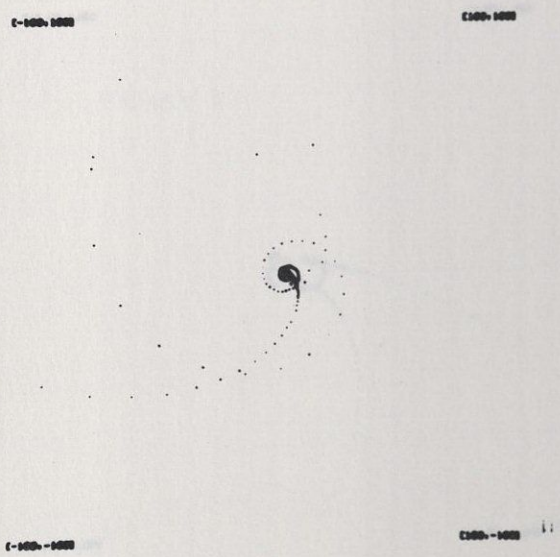


Figure 10

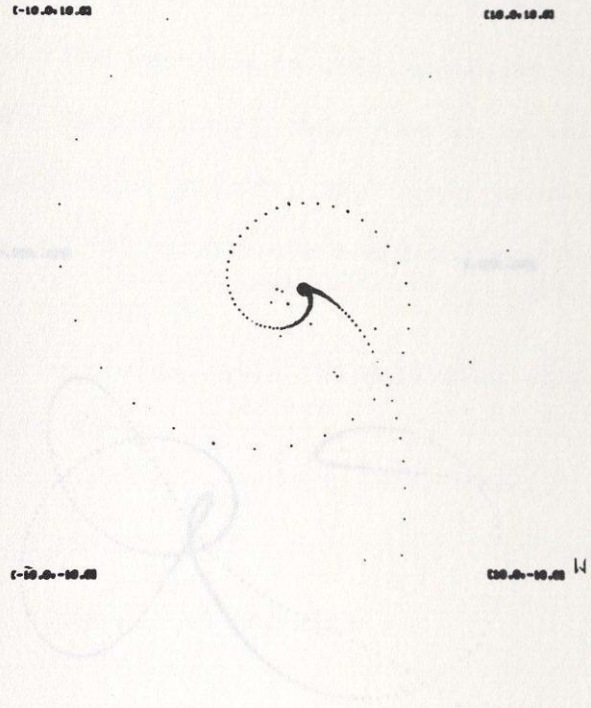


Figure 11

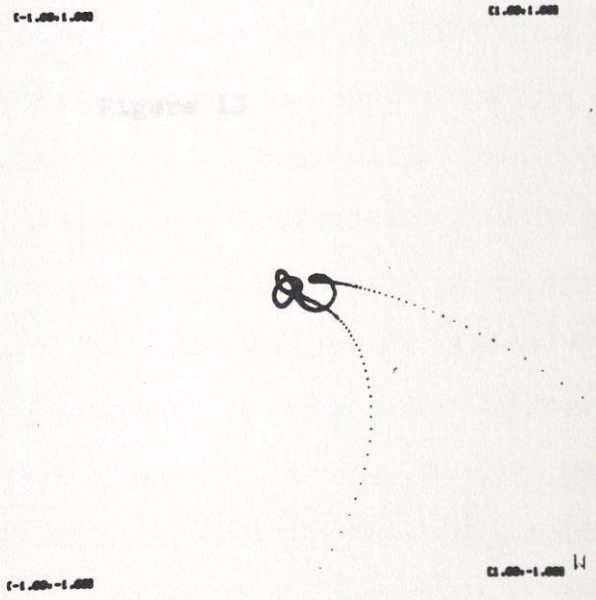


Figure 12

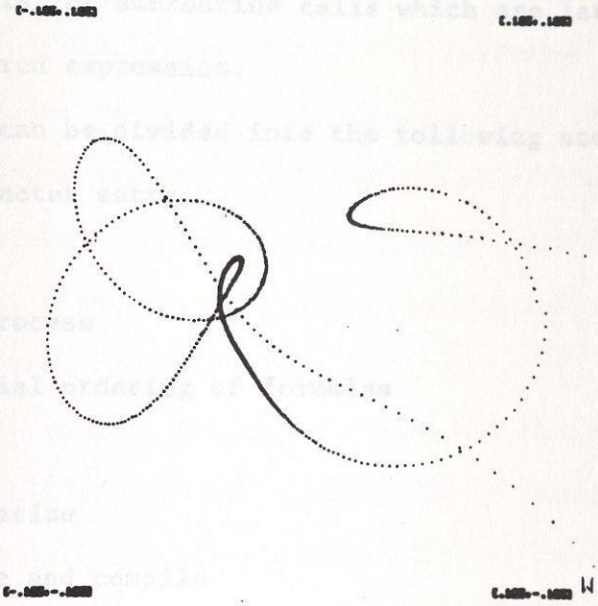


Figure 13

algorithm is used which he gives for the special case of algebraic mathematical notation. The key step in this procedure is the ordering of the input characters into a string which can be parsed by a context-free simple precedence grammar [9]. After parsing, another routine creates a list of subroutine calls which are later executed to compute the desired expression.

The algorithm can be divided into the following steps:

i) Character entry

ii) Sort

iii) Preprocess

iv) Partial ordering of formulas

v) Sort

vi) Linearize

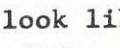
vii) Parse and compile

i) Character entry

The character recognizer included in SHAPESHIFTER was written by Kenneth Ledeen at Harvard University [6]. With a training program the user teaches the computer his own handwriting; this information is punched on paper tape and is read into SHAPESHIFTER at run time. In effect each user has a version of the character recognizer tailored to his own writing style. The particular scheme by which characters are recognized is of no consequence to the parsing algorithm. The recognizer must supply five pieces of information: the name of the character and its four descriptive co-ordinates, the x minimum, x maximum, y minimum, and y maximum of the area where it was written

on the tablet.

After a character has been printed and recognized, a canonical form of the character, in the same position and of the same dimensions as the printed character, replaces the "ink" on the scope. This both neatens the display and tells the user how the written character was interpreted. The character's name and its co-ordinates are stored in a table. Characters are erased by writing a special "scrub" mark over them; this deletes them from both the display file and the character table. Because no part of the parsing algorithm uses time sequence information, characters may be printed in any order and may be scrubbed at any time.

If more than one formula is printed, each but the last must be terminated by a semicolon. When a semicolon is recognized a horizontal line is displayed at its y minimum. (See figure 8. The semicolon and line together look like this: ) The horizontal lines extending across the screen at the bottoms of the semicolons divide the characters into groups; each formula must be composed of exactly one group of characters.

The input vocabulary is described in Appendix 2.

ii) Sort

The rows of the input character table are re-ordered according to the x minimum of the characters, from smallest to largest. (N.B. Left-to-right is the direction of increasing x.) The purpose of this step is to put strings of characters (such as "SIN") in order so that they can be recognized by the preprocessor.

iii) Preprocess

The preprocessor is a two-dimensional lexical analyzer. In principle it could be omitted entirely; practically, however, it makes possible a reduction in the size of the terminal vocabulary and syntax of the string parser.

In formal terms, the preprocessing is a many-one mapping from sets of one or more characters in the input character table to terminal symbol tokens. The word "token" is used to indicate that descriptive co-ordinates are associated with each symbol.

Typically, the preprocessor reduces each valid string of digits and decimal points to a "syntactic category," i.e. terminal symbol, called NUMBER, and reduces each string of letters naming a trigonometric function to a syntactic category called TRIGNAME. The changes are completely described in Appendix 3.

iv) Partial ordering of formulas

If more than one formula was written, this step places them in an order in which they can be computed, provided that such an order exists. The basis of the reordering is that a variable must be defined on the left-hand side of an equation before it can appear on the right-hand side of another equation; i.e. the value of a variable must have been computed before it can be used to compute the value of another variable. The result is to augment the x co-ordinates of the characters in some of the formulas so that the formulas are strung out from left to right according to the sequence in which they must be computed. Only the internally used co-ordinates are altered; that is, the appearance of the display does not change. The algorithm for this step follows from the formalism in Appendix 6.

v) Sort

The rows of the table must now be resorted, as in (ii), if they were disordered in step (iv). The input character table is now in the form required by the linearization procedure.

vi) Linearization

The object of this step is to linearize the two-dimensional mathematical notation into an equivalent one-dimensional notation; i.e., to generate from the set of terminal symbol tokens the ordered string of terminal symbols (without co-ordinates) which is the input to the simple precedence parser. The terminal symbols in this string include the syntactic categories defined by the preprocessor and additional symbols generated as the two-dimensional structure of the algebraic expression is analyzed.

The linearizer is the heart of Anderson's efficient recognition algorithm. So that this algorithm may be used, certain restrictions are made on the formation of expressions; for example, the divisor and dividend of a quotient must fall between the ends of the division bar in the x-direction. In addition, the specific language implemented in SHAPESHIFTER allows only three types of character configurations which violate the essential linearity of the language, namely the use of the bar to indicate division, the radical sign to indicate root-taking, and superscripting to indicate exponentiation. In his dissertation Anderson completely defines the linearization algorithm; the steps in the procedure for this restricted grammar form a proper subset of those given by him, except for the trivial exception noted

in Appendix 5.

Appendix 4 describes a detail of the implementation of this step.

vii) Parsing

For this implementation Anderson developed a grammar which is simple precedence and has only twenty-two syntax rules (Appendix 5). Strings with this syntax can be parsed by a well-known technique.

§7. Details of Implementation

i) Machine facilities used

This program runs on a DEC PDP-1 central processor in the Division of Engineering and Applied Physics. The machine is equipped with 24K words of 18 bit core memory and a 131K word drum (DEC type 24). Also attached is the display (a DEC type 340 with three slave scopes), the Grafacon, an Adage A/D converter to which the joystick is attached, and assorted switches which can be read by the program.

ii) Size of the program

The storage allotment is approximately as follows: 12K words of PDP-1 program and 4K words of display file are permanently core-resident; 8K words of program (the character recognizer and parser) and 8K words of data buffer, which occupy the same section of momory, are swapped at various stages of the execution. The program is written in DECAL, a combination assembler and low-level compiler.

iii) Parsing and computation times

Some typical expressions, the time required to parse them, and

the time required to compute their values for the 1024 points in the domain are given below.

Expression	Parsing Time	Computation Time
$W = Z^2$	.11 sec.	15 sec.
$W = \sqrt{Z}$	.10 sec.	21 sec.
$W = Z + \frac{1}{Z}$	.19 sec.	14 sec.
$W = \text{SIN } Z$	.08 sec.	64 sec.
$W = \frac{1+Z-Z^2}{Z - \frac{1}{Z}}$	.49 sec.	48 sec.
$W = \frac{Z}{Z - \frac{Z}{Z - \frac{1}{Z}}}$	.44 sec.	66 sec.

These parsing times are a twentieth to a tenth the times required for the parsing of similar expressions by Anderson's general algorithm (implemented on faster machines [2, pp. 88-9]); they are in the same range as the times given by Martin [7].

§8. Conclusion

Both the difficulty of writing this program and its restricted usefulness as a tool for teaching or experimenting with mathematics are largely due to characteristics of the machine on which it is implemented. The PDP-1 is slow by contemporary standards. The absence of floating-

point hardware complicated the coding process and makes computation times unnecessarily long, whereas the parsing, which is for the most part a logical operation, is done with adequate speed. The time required to compute the range of a function is about one hundred times as long as the time taken for parsing its two-dimensional formula. Insufficient memory is also an annoyance which slows execution, since portions of the program reside in secondary storage. This implementation nevertheless demonstrates the practicality of two-dimensional input for this type of mathematical application.

The main lesson to be learned from this project is that the parts of the program which one might have expected to be the most difficult to implement and the most costly in terms of computing requirements -- namely, the recognition procedures -- are fairly easy to write, small in size, and rapid in operation. Moreover, they are essentially modular; this implementation of these routines could be used in many other programs. Finally, they do not need to be written in any special programming language. While a list-processing language might have been useful for implementing the two-dimensional parsing algorithm -- particularly the string parser -- the compiler would have produced less efficienct code and might finally have been a disadvantage. Execution of the entire parser as coded is so rapid that it is hardly noticed in the interactive environment.

This system, then, is not essentially complex. Nor should it be regarded as merely an experimental toy: several practical applications were noted in §1 and others can easily be imagined. In the future,

when the requisite hardware and software are more readily accessible, similar systems will be implemented as a matter of course.

Part II. Recognition of Certain Two-Dimensional Programming Languages

§1. Purpose

Certain general programming constructs are inherently two-dimensional. Program listings are often marked with braces and parentheses to suggest that a series of statements is considered as a single statement for some purpose. For example, such a statement block might be the interior of an iteration loop. The blocks themselves may be grouped or nested. One-dimensional languages indicate such groupings by artificial conventions, such as "DO n" in FORTRAN and "begin" and "end" in ALGOL. A two-dimensional language, however, could show the groupings by the same graphical devices that programmers commonly use on paper (figure 14).

An algorithm for recognizing blocks of statements set off by braces is general in two senses. First, the syntax of an individual statement is irrelevant; the only assumptions are that statements are are written one below another like lines of code, and that a "grouping symbol" bounds the block of statements in the vertical direction. Hence the algorithm is input language independent. Second, the recognition algorithm need only determine the extent of the groupings; the use made of this information is not part of the algorithm. The input could be translated into FORTRAN of ALGOL or compiled directly into machine language. Hence the algorithm is output language independent.

This section describes such a recognition algorithm. For an implementation, an algebraic language was selected as the input, and a string language similar to ALGOL is used for writing the output. The

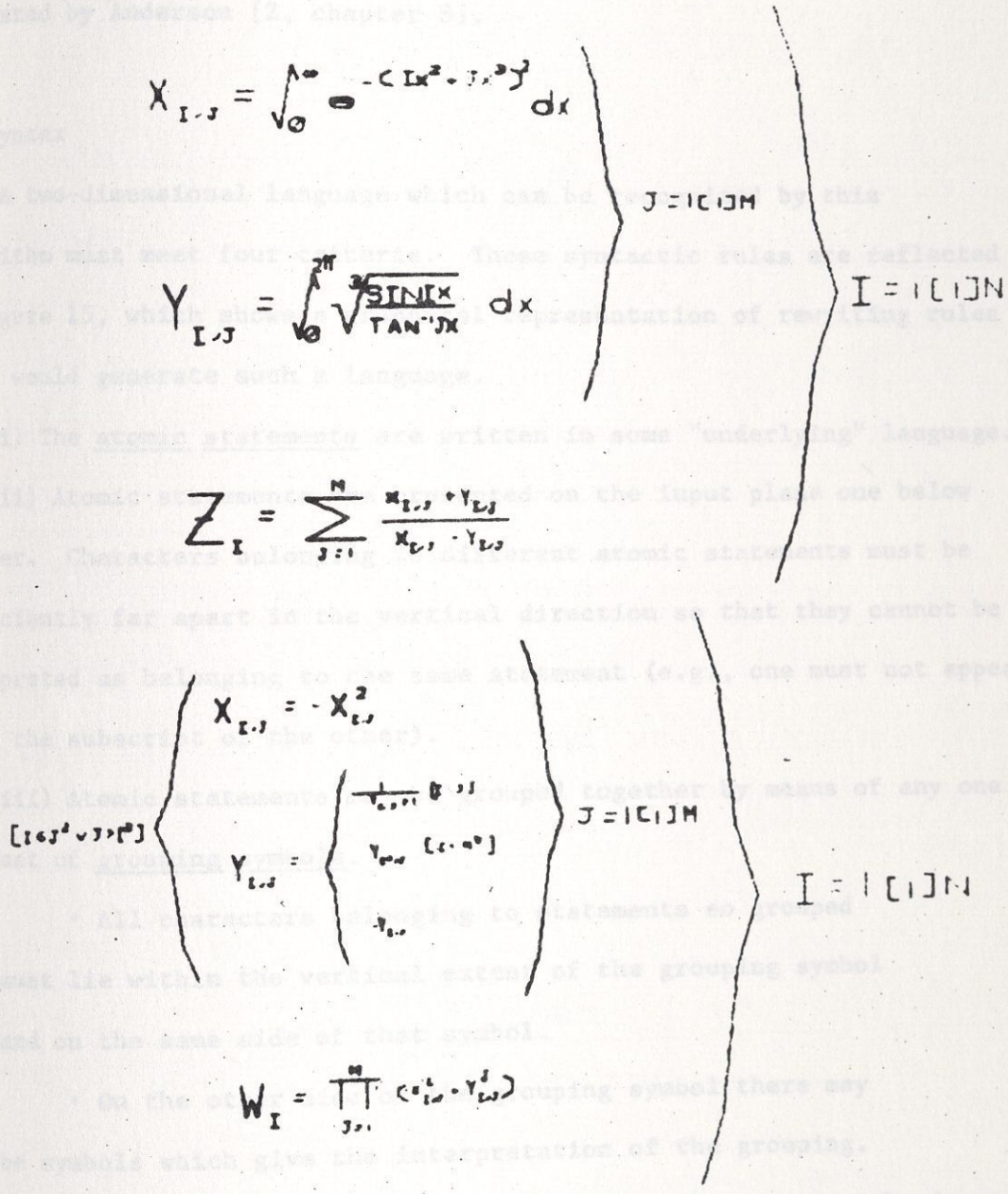


Figure 14

same algorithm, however, could be used for other inputs and outputs. The languages which can be recognized by this algorithm were originally suggested by Anderson [2, chapter 8].

§2. Syntax

A two-dimensional language which can be recognized by this algorithm must meet four criteria. These syntactic rules are reflected in figure 15, which shows a graphical representation of rewriting rules which would generate such a language.

i) The atomic statements are written in some "underlying" language.

ii) Atomic statements are presented on the input plane one below another. Characters belonging to different atomic statements must be sufficiently far apart in the vertical direction so that they cannot be interpreted as belonging to the same statement (e.g., one must not appear to be the subscript of the other).

iii) Atomic statements may be grouped together by means of any one of a set of grouping symbols.

- All characters belonging to statements so grouped must lie within the vertical extent of the grouping symbol and on the same side of that symbol.
- On the other side of the grouping symbol there may be symbols which give the interpretation of the grouping.

The set including a grouping symbol and all the characters on the input plane associated with it in either of these ways is a grouped statement. A statement is an atomic or grouped statement.

iv) The word "statement" may replace the words "atomic statement" in rules (ii) and (iii).

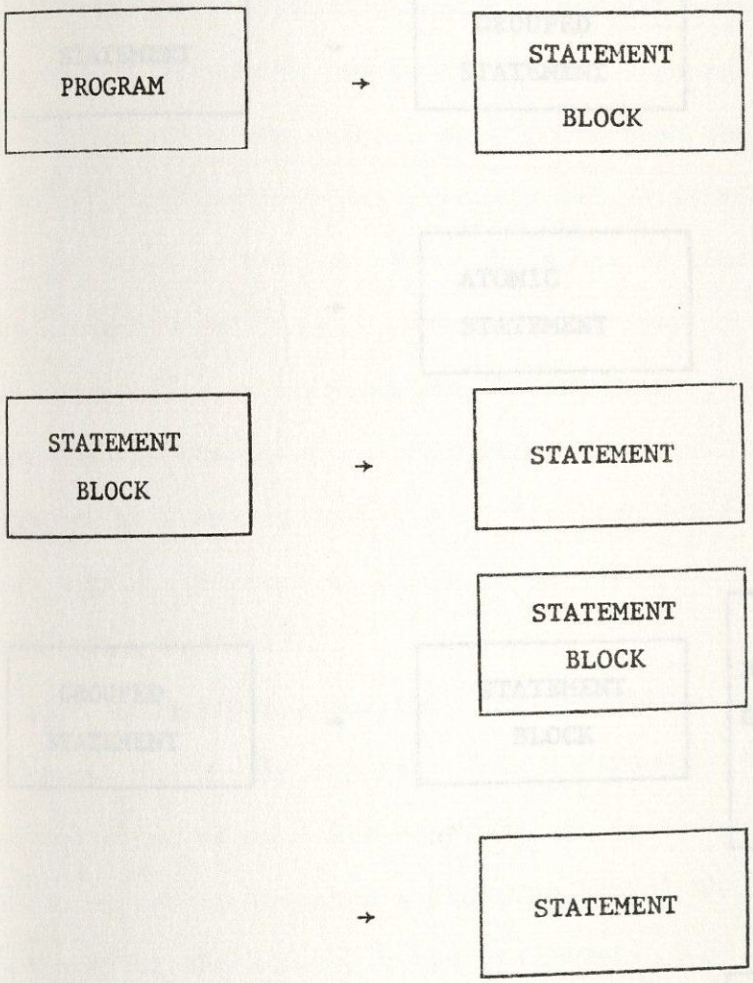
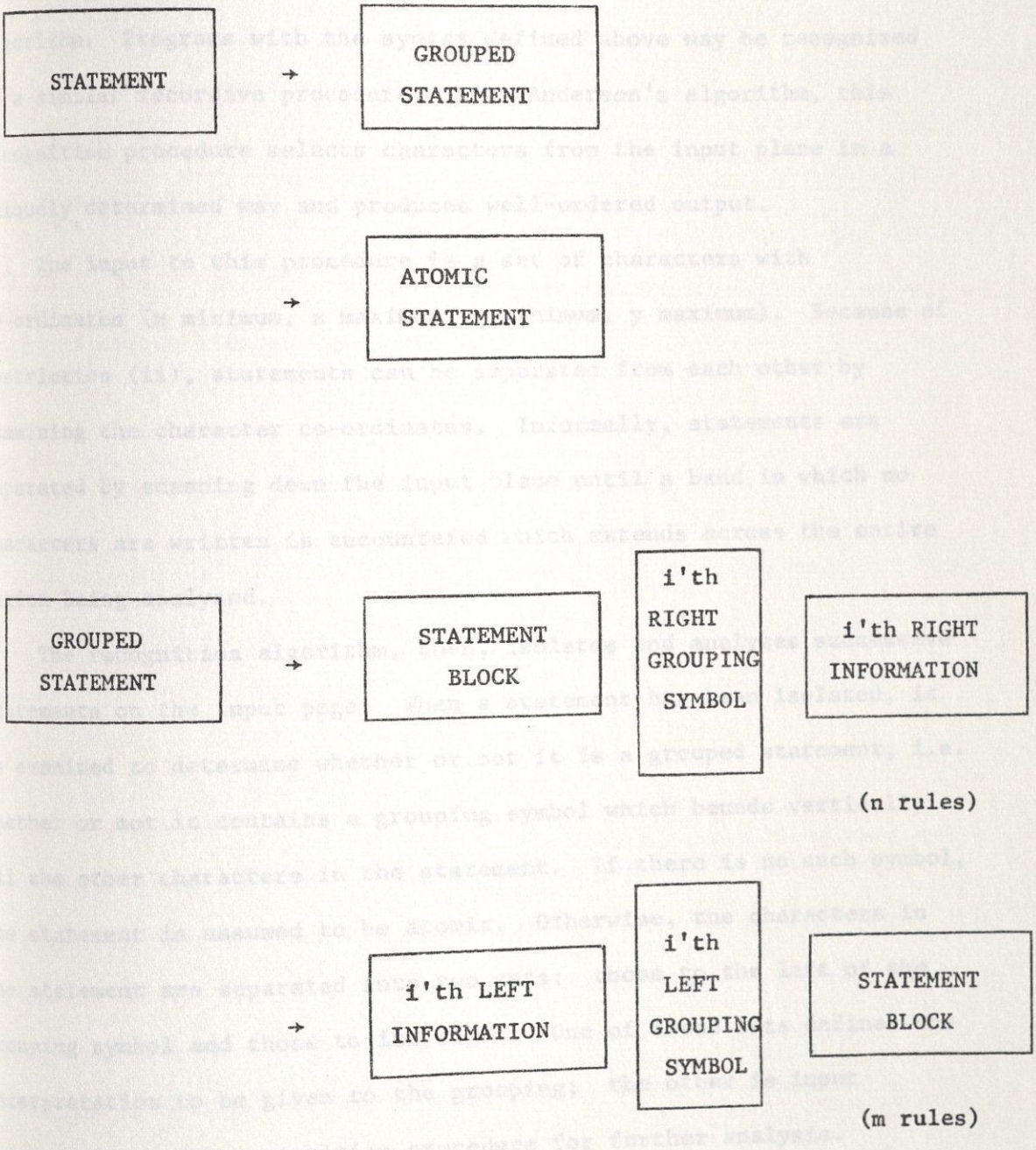


Figure 15

(Figure continued on next page)


STATEMENT
GROUPED STATEMENT
ATOMIC STATEMENT
GROUPED STATEMENT
STATEMENT BLOCK
i'th RIGHT GROUPING SYMBOL
i'th RIGHT INFORMATION
(n rules)
i'th LEFT INFORMATION
i'th LEFT GROUPING SYMBOL
STATEMENT BLOCK
(m rules)

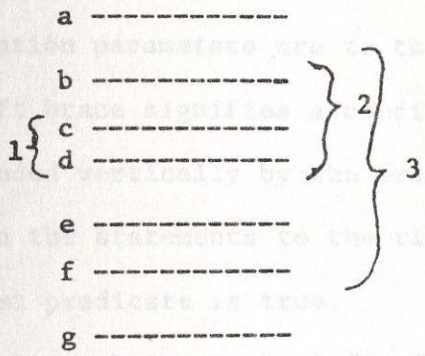
§3. Recognition Algorithm

Anderson [2, Appendix 1] shows how certain two-dimensional mathematical expressions can be recognized by an algorithm much simpler and more efficient than his general syntax-directed recognition algorithm. Programs with the syntax defined above may be recognized by a similar recursive procedure. Like Anderson's algorithm, this recognition procedure selects characters from the input plane in a uniquely determined way and produces well-ordered output.

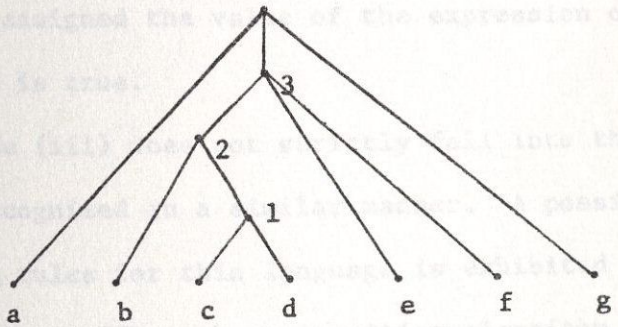
The input to this procedure is a set of characters with co-ordinates (x minimum, x maximum, y minimum, y maximum). Because of restriction (ii), statements can be separated from each other by examining the character co-ordinates. Informally, statements are separated by scanning down the input plane until a band in which no characters are written is encountered which extends across the entire region being analyzed.

The recognition algorithm, then, isolates and analyzes successive statements on the input page. When a statement has been isolated, it is examined to determine whether or not it is a grouped statement, i.e. whether or not it contains a grouping symbol which bounds vertically all the other characters in the statement. If there is no such symbol, the statement is assumed to be atomic. Otherwise, the characters in the statement are separated into two sets: those to the left of the grouping symbol and those to its right. One of these sets defines the interpretation to be given to the grouping; the other is input recursively to the recognition procedure for further analysis.

The output from the recognition algorithm can be pictured as a tree structure. The leaves of the tree are atomic statements; the nodes are grouped statements. For example, a program of the form:



would be represented as the tree:



§4. Implementation

For the purposes of implementation on the DEAP's PDP-1 computer, an algebraic language was designed which has only one type of atomic statement: assignment, i.e. "<variable>=<expression>." Mathematical expressions are written in textbook form. The rules governing their formation and recognition are essentially as given by Anderson.

Three types of grouped statements are recognized by this program (cf. figures 14, 16, and 17):

i) A right brace defines an iteration loop. The statements to the left of the brace and bounded by it vertically are the interior of the loop; the iteration parameters are to the right of the brace.

ii) A left brace signifies a conditional statement. If the characters bounded vertically by the brace and to its left form a boolean predicate, then the statements to the right of the brace are to be executed if that predicate is true.

iii) If the symbols to the left of a left brace are of the form "<variable>=", then the symbols to the right of the brace are interpreted as a series of expressions paired with boolean predicates. The variable is to be assigned the value of the expression of which the associated predicate is true.

Case (iii) does not strictly fall into the schemata defined above, but is recognized in a similar manner. A possible set of two-dimensional rewriting rules for this language is exhibited in Appendix 8. Appendix 9 completely specifies the recognition algorithm used in this implementation.

Rather than compiling statements into executable machine code, this program rewrites them as equivalent statements in a one-dimensional language similar to ALGOL. Figure 18 shows the output program corresponding to the input shown in figures 14, 16, and 17.

As the linear character string is generated from the two-dimensional input, a single procedure is used to add characters to the output string. If a string were being prepared for parsing, this routine would append single characters to the right-hand end of the string. In this program,

$$X_{I,J} = \int_0^{\infty} e^{-(Ix^2 + Jx^3)^2} dx$$

$$Y_{I,J} = \int_0^{2\pi} \sqrt[3]{\frac{SINIX}{TAN^{-1}JX}}\, dx$$

$J = 1(1)M$

$$Z_I = \sum_{J=1}^{M} \frac{X_{I,J} + Y_{I,J}}{X_{I,J} - Y_{I,J}}$$

Figure 16

$$X_{I,J} = -X^2_{I,J}$$

$$Y_{I,J} = \begin{cases} \frac{1}{Y_{I,J}+1} & [I<J] \\ Y_{I^2,J} & [I<N^{\frac{1}{2}}] \\ -Y_{I,J} & \end{cases} \quad J = 1[1]M$$

$$W_I = \prod_{J=1}^{M} \left(X^I_{I,J} + Y^J_{I,J}\right)$$

Figure 17

```
for I=1 step 1 until N do begin:
     for J=1 step 1 until M do begin:
          X↓(I,J)= integral (0 to  infinity ) e↑[-(IX↑[2]+JX↑[3])↑[2]] dX:
          Y↓(I,J)= integral (0 to 2pi) root(3,(sinIX)/(arctanJX)) dX:
     end:
     Z↓(I)= sum (J=1 to M) (X↓(I,J)+Y↓(I,J))/(X↓(I,J)-Y↓(I,J)):
end:
for I=1 step 1 until N do begin:
     for J=1 step 1 until M do begin:
          if I leq Jf[2] or J gt If[2] then  begin:
               X↓(I,J)=-X↓(I,J)↑[2]:
               if I lt J then Y↓(I,J)=(1)/(Y↓(I,J)+1):
                    else if I lt Mf[(1)/(2)] then Y↓(I,J)=Y↓(If[2],J):
                    else Y↓(I,J)=-Y↓(I,J):
          end:
     end:
     W↓(I)= product (J=1 to M) (X↓(I,J)↑[I]+Y↓(I,J)↑[J]):
end:
```

Figure 18

however, the output from the linearizer is instead placed in a display file, so the procedure is designed to improve the readability of the displayed text. It inserts carraige returns, tabs, and semicolons when called with certain arguments and makes use of both upper and lower case (lower case letters are for the most part the boldface letters in ALGOL). Some special characters are displayed, for example "≠" as "neq" . The resulting output program is formally unambiguous.

The entire recognition procedure is quite fast; the translation time for the illustrated input program was about 20 seconds.

§5. Program Windowing

The cathode-ray tubes used with computers generally have dimensions between five and twenty inches square. If the computer memory contains a representation of a two-dimensional surface, this size limitation would restrict the amount of information which could conveniently be manipulated in a graphical computer system unless the information is handled selectively. The system can be used to represent far more information by the familiar technique of windowing, i.e., by displaying at any time only a portion of the information in the computer memory.

A two-dimensional programming system requires a windowing technique similar to that commonly used for the selective display of pictorial data. The window must be locatable at any place on the input page and must change size so as to magnify or shrink the input characters.

By means of magnifying windows the implemented program enables the user to inspect various portions of the page on which he is writing. (Figures 16 and 17 are windows on the program in figure 14.) If a

user wishes to write a complicated expression in a region which is small in the current window, he may magnify the page so that the small region fills the entire area of the scope. Anything now written will be stored internally according to the new page co-ordinates of the tablet and scope. When the window is restored to its previous size, the complicated expression will seem to have been written in tiny handwriting. Because the points on the tablet are kept in one-to-one correspondence to the points on the scope, the tablet surface need be large no more than the scope need be large; both are mapped into a portion of the "real" input-output surface.

§6. Conclusion

The recognition algorithm described in this section applies only to a restricted class of possible two-dimensional programming languages. The simplicity and speed with which one of these can be recognized, however, opens the way to experimentation in the design of other languages using graphical input. Since important program modules were already available, the implementation of this system took only a short time; with this base other variations could now be explored.

The feasibility of a programming system with two-dimensional input is, therefore, amply demonstrated. But little thought has been given to defining what logical constructs can be represented two-dimensionally, and how they might be represented so that they could be recognized by a simple procedure. Co-routines and parallel processes, for example, could be indicated by some convenient graphical device. Now that two-dimensional syntactic analysis is known to be reducible to a

one-dimensional analysis in certain important cases, further generalizations of these techniques and algorithms can follow quickly.

Conclusion

In terms of special hardware requirements, the problem of two-dimensional input is fairly complex. Convenient two-dimensional input requires at present the dedication of an entire computer equipped with a tablet and scope, which are expensive peripheral devices. To enable the general use of two-dimensional input, the cost of these devices must be lowered, and the computer itself must come to bear -- perhaps through timesharing, perhaps through some other technique -- less of the burden of trivial data input.

The two programs described in this thesis demonstrate that neither dynamic character recognition nor the recognition of two-dimensional algebraic expressions is as difficult as was believed a few years ago. Machine-independent specifications for both procedures are available and are flexible enough to permit experimentation with extensions and adaptations. Even on a machine as slow and restrictive as the PDP-1, these procedures are fast enough to be useful in an interactive system.

Those who implement these algorithms in the future should keep in mind their essential generality. Good recognition procedures will disassociate the logical structure of the algorithms from any particular set of inputs and outputs. These modular procedures should be skeletons onto which a specific character set or syntax may be added merely by describing it in some formal or informal way. The process of programming

a system such as the ones described here ought to be synthetic: the programmer should be able to deal with only those aspects of the system not common to other similar programs.

Appendix 1. SHAPESHIFTER: Description of States

Figure 19 is a state-diagram of SHAPESHIFTER. The circles represent states; the boxes represent buttons. An arrow leading from one circle to another indicates that the progrma will enter the latter state immediately upon completion of the operation associated with the former state. If an arrow leads from a circle to a box, the circle stands for a "wait state"; the operation associated with will continue until a button is pressed. The arrow leading from a box points to the state which the program enters after that button has been pressed.

State 1 is the ground state in which the program begins; no formulas are displayed in this state. Either of the two "standard" domains may be recreated (2,3), the tree structure for the character recognizer may be read in off paper tape (4), and the pretranslation parameter α may be changed (5). When the ENTER button is pressed the program goes into the character entry phase (6). From this state the user may erase the formulas and return to the ground state (ABORT, 7) or go on to parse the character configuration that he has written. The parsing procedures have two exits. If the characters are determined not to constitute a valid set of formulas (10) a wait state is entered from which the program may re-enter the entry phase (EDIT, 11) or the ground state (ABORT). If the formulas are judged to be legal the progrma enters the execution phase (12) in which the value of the transformation is computed for each point in the domain and is plotted in the range. States 15, 17, and 18 are analogous to states 5, 2, and 3 respectively. The

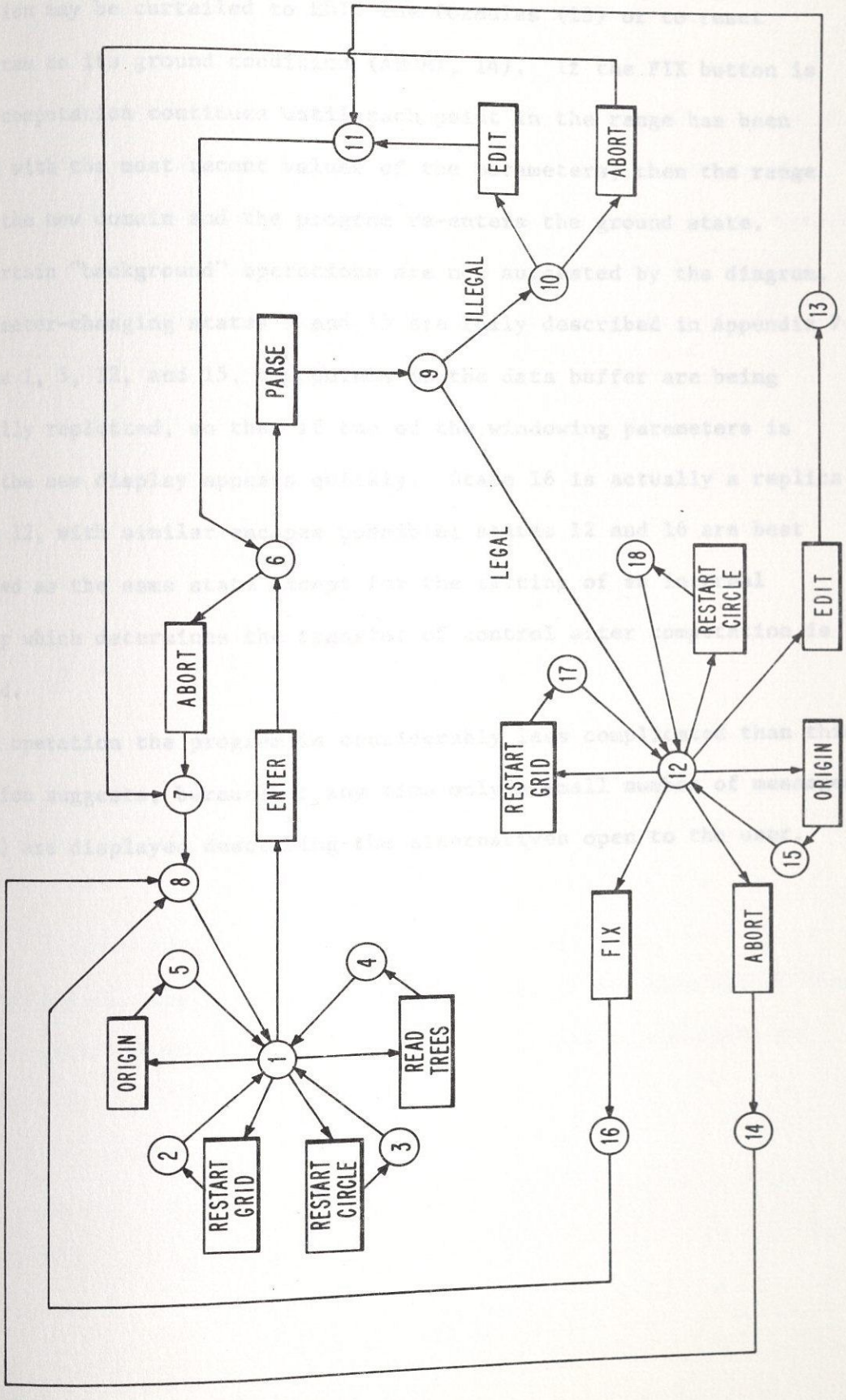

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
ORIGIN
RESTART GRID
RESTART CIRCLE
READ TREES
ENTER
ABORT
PARSE
ILLEGAL
LEGAL
EDIT
ABORT
RESTART GRID
RESTART CIRCLE
EDIT
ORIGIN
FIX
ABORT

Figure 19

computation may be curtailed to EDIT the formulas (13) or to reset the program to its ground condition (ABORT, 14). If the FIX button is pushed, computation continues until each point in the range has been computed with the most recent values of the parameters; then the range becomes the new domain and the program re-enters the ground state.

Certain "background" operations are not suggested by the diagram. The parameter-changing states 5 and 15 are fully described in Appendix 7. In states 1, 5, 12, and 15, the points in the data buffer are being continually replotted, so that if one of the windowing parameters is changed the new display appears quickly. State 16 is actually a replica of state 12, with similar escapes possible; states 12 and 16 are best considered as the same state except for the setting of an internal flip-flop which determines the transfer of control after computation is completed.

In operation the program is considerably less complicated than this description suggests, because at any time only a small number of messages (buttons) are displayed describing the alternatives open to the user.

Appendix 2. Input Vocabularies

One of the several advantages of a "learning" character recognizer such as the one incorporated in these programs is that it permits the specification of a vocabulary consisting of exactly those special symbols needed for a particular application. The vocabulary used for writing the formulas in SHAPESHIFTER consists of the following characters:

Roman upper-case alphabet, A - Z

Digits, 0 - 9

. (decimal point)

i

e

π

√‾‾

=

+

- (minus sign or quotient bar)

;

(scrub mark)

The vocabulary for the program described in Part II contains all of these characters with the exceptions of "i" and ";" , and in addition the following:

^

<

$\leq$ (may also be written as $\not>$)

>

$\geq$ (may also be written as $\not<$)

∞

}

{

Π

Σ

$\int$

$\neq$

d

,

]

[

Associated with each printed character is the rectangle defined by its minimum and maximum x and y co-ordinates. The canonical form of the character which appears after it has been recognized fills this rectangle.

Three of these characters are special. The scrub mark erases all those characters of which the centers are within the rectangle associated with the scrub. Erasures may be either selective or general -- either a single character or an entire expression can be deleted with one scrub mark.

The radical sign, though written as a single character, is displayed as two pieces: $\surd$ and $\overline{}$. Each part has its own proportions and fills

its own rectangle. The character recognizer extracts additional coordinates for this character, namely those of the point of juncture between the two pieces.

In SHAPESHIFTER, the semicolon is always displayed with a separation line (figure 8).

Appendix 3. Preprocessor

Like a lexical analyzer, the preprocessor has no formal status in a description of the recognition algorithm; by adding rules to the grammar it would theoretically be possible to eliminate this step entirely. Practically, however, the application of a few non-formal, ad hoc techniques at this stage permits the rest of the parsing algorithm to be much simpler.

The preprocessor searches the input plane in the left-to-right direction, looking for special characters and character strings. Precisely, it makes the following changes: (Some of these are omitted in the program described in part II, since that program does not need the information which the preprocessor creates for the purpose of compiling executable machine code.)

a) Reduce the following strings to a syntactic category called TRIGNAME:

SIN COS TAN SEC CSC COT

SINH COSH TANH SECH CSCH COTH

SIN^{-1} COS^{-1} TAN^{-1} SEC^{-1} CSC^{-1} COT^{-1}

$SINH^{-1}$ $COSH^{-1}$ $TANH^{-1}$ $SECH^{-1}$ $CSCH^{-1}$ $COTH^{-1}$

b) Reduce the following strings to a syntactic category called FUNCOP:

LOG EXP

c) Distinguish between minus sign and quotient bar.

d) Reduce plus and minus signs to a syntactic category called ADDOP.

e) Reduce Σ and Π to a syntactic category called SUMOP.

f) Reduce valid strings of digits and decimal points to a syntactic category called NUMBER.

g) Assign the syntactic category VARIABLE to each remaining alphabetic character.

h) Assign syntactic categories in a one-one manner to the remaining characters.

i) The preprocessor pairs each syntactic category which it has created with the number of the row in which it appears. These category-row number pairs are the terminal symbol tokens (terminal symbols with co-ordinates) which are passed to the rest of the parser.

A few notes about these changes:

a, b, and f: If a valid string is isolated in the two-dimensional input plane, the rows corresponding to the characters in the string are removed from the input character table and replaced with a single row containing the co-ordinates of the entire string.

c: Minus sign and quotient bar are distinguished as follows: Define a square bisected by the character, for example,

If there is a character which is in part contained in the upper rectangle, and there is also a character which is in part contained in the lower rectangle, then the character is a bar; otherwise, a minus sign. This distinction could also be made by the linearizer.

a, b, d, e, f, and g: A name is entered into the row which identifies the operator, variable, or number as a specific token of its syntactic category. In the final step the compiler can then return to the original character table to determine the particular instance of a category for which the symbol is a token; for example, which trigonometric function is being used, or where the value of a given number or constant is stored.

Appendix 4. Rule for determining Superscripts and Subscripts

Anderson states the following rule for determining whether or not a character is the superscript or subscript of another [2, p. 124]:

"R6: all characters in a subscript or superscript must be sufficiently below or above (respectively) the center of the letter or factor to which they apply so that they are not on the same typographical line as that letter or factor."

The word "sufficiently" cannot be defined in any elegant way.

i) Anderson later suggests that the characters "on the same typographical line as" some character c_1 are all those characters c_2 such that $|ycent(c_1) - ycent(c_2)| < htol$. Here htol is a constant of the implementation, or perhaps is adjustable by the user. The characters printed in mathematical expressions, however, vary considerably in size, and the tolerance must therefore vary as well. In an expression such as A^{2^3}, for example, the vertical distance from the "2" to the "3" would not be sufficient to make "2^3" the exponent of "A".

ii) An alternative rule might be that $|ycent(c_1) - ycent(c_2)| < htol$, where $htol = p * (ymax(c_1) - ymin(c_1))$, and p is some quantity between 0 and 1. This is a better suggestion, but has the disadvantage that certain symbols (such as "_" and "=") have height which is disproportionate to their true size.

The following rule is used in these programs: Let c_1 and c_2 be candidates for base and superscript or subscript, respectively, and assume that the six co-ordinates, xmin, xcent, xmax, ymin, ycent, and

ymax, are available for each. Then

```
htol ← if c1 = horizline then xmax(c1) - xmin(c1) ;
     else if c1 = plussign or c1 = minussign or
          c1 = equalsign then 2*(xmax(c1) - xmin(c1)) ;
     else (ymax(c1) - ymin(c1))/3 ;
if abs(ycent(c1) - ycent(c2)) < htol then goto sameline else
     goto differentline ;
```

For most cases this rule reduces to (ii). The value of p is $\frac{1}{3}$; this seems to conform to general usage.

Appendix 5. SHAPESHIFTER: Syntax for Linearized Mathematics

• Terminal Symbols:

LEFTEND RIGHTEND , / √ ↑ ⇑ ; = ADDOP FUNCOP TRIGNAME VARIABLE

[ ] ⟦ ⟧ ( )

• Nonterminal Symbols:

GOAL ASSIGNS EXPRESSION EXPRESSION1 TERM TERM1 MULTERM TRIGTERM FACTOR

• Syntax Rules:

1 <GOAL> → <LEFTEND> <ASSIGNS> <RIGHTEND>

2 <ASSIGNS> → <ASSIGNS> ; <VARIABLE> = <EXPRESSION>
3 → <VARIABLE> = <EXPRESSION>

4 <EXPRESSION> → <EXPRESSION1>

5 <EXPRESSION1> → <EXPRESSION1> <ADDOP> <TERM>
6 → <ADDOP> <TERM>
7 → <TERM>

8 <TERM> → <TERM1>

9 <TERM1> → <TERM1> <MULTERM>
10 → <MULTERM>

11 <MULTERM> → <TRIGTERM>
12 → ⇑⟦<EXPRESSION>⟧ <TRIGTERM>
13 → <FUNCOP> <MULTERM>
14 → (<EXPRESSION>)/(<EXPRESSION>)
15 → <FACTOR>

16 <TRIGTERM> → <TRIGNAME> <MULTERM>

17 <FACTOR> → √[<EXPRESSION>,<EXPRESSION>]
18 → √[,<EXPRESSION>]
19 → (<EXPRESSION>)
20 → <FACTOR>↑[<EXPRESSION>]
21 → <VARIABLE>
22 → <NUMBER>

This grammar is a proper subset of that given by Anderson, except for a few rules:

1 The symbols "LEFTEND" and "RIGHTEND" delimit the string to be parsed. This rule merely tells the parser when to stop.

2,3 These rules handle multiple assignment statements.

12 Exponents of trigonometric names are linearized in this special way to simplify the operation of the parser.

Below is a precedence matrix for this restricted grammar.

	LEFTEND	RIGHTEND	,	/	↑	↑	ADDOP	FUNCOP	NUMBER	TRIGNAME	VARIABLE	[	]	⊔	⊓	(	)	EXPRESSION	TERM	TRIGTERM	MULTERM	FACTOR	GOAL	TERM1	EXPRESSION1	ASSIGNS	;	=
ASSIGNS																												
EXPRESSION1																												
TERM1																												
GOAL																												
FACTOR																												
MULTERM																												
TRIGTERM																												
TERM																												
EXPRESSION																												
)																												
(																												
⊓																												
⊔																												
]																												
[																												
VARIABLE																												
TRIGNAME																												
NUMBER																												
FUNCOP																												
ADDOP																												
↑																												
↑																												
/																												
+																												
,																												
RIGHTEND																												
LEFTEND																												

Appendix 6. SHAPESHIFTER: Partial Ordering of Formulas

If the input consists of more than one formula, the formulas must be placed in an order in which they may be successively computed. This appendix describes the formal properties of such a re-ordering.

Formulas are defined by semicolons. More precisely, if there are n-1 semicolons in the input character set these semicolons determine n regions in the input plane. The characters in these n regions are defined to constitute n formulas, $F_1, F_2, \ldots, F_n$. Let the leftmost character in F_i be V_i; if V_i is not the name of a variable (i.e. a Roman upper-case letter), the formulas are illegal. Let R_i be the set consisting of the variables in F_i except V_i. Then the partial ordering $<$ between the F_i is defined as follows, where p is a permutation of the integers $\{1, 2, \ldots, n\}$:

$$F_{p(1)} < F_{p(2)} < \ldots < F_{p(n)} \quad \text{if and only if}$$

i) $R_{p(1)} = \{Z\}$

ii) $V_{p(n)} = W$

iii) $R_{p(q)} \subseteq \{V_{p(1)}, V_{p(2)}, \ldots, V_{p(q-1)}, Z\}$ for $q = 2,3, \ldots, n$.

If no such ordering can be found, the input characters do not form a valid sequence of formulas. If such an ordering is found, (i-1) times the width of the tablet is added to the x-coordinates of all characters in $F_{p(i)}$ for $i = 2, \ldots, n$; this has the effect of creating a virtual "line" with the formulas in the correct order from left to right. Note that the following are valid sequences of formulas:

$W = A^2;$

$A = Z + Z^2$

$A = Z^2 + A;$

$A = \text{SIN } Z + Z;$

$W = A^3 + Z^2$

Appendix 7. SHAPESHIFTER: Operating Instructions

1. Where to find the program

The binary of SHAPESHIFTER is on Dectapes 168 and 180, and is called LEWIS BINARY. All the DS, DM, DL, LDSYM, and LM files are on Dectape 168.

In addition, a set of trees for the character recognizer (which encode the user's handwriting) is needed to operate the program. Trees for my own handwriting, punched on paper tape, are available in the paper tape rack and are labelled "The Best Trees Ever." A user who wishes to produce his own trees should use the version of the character recognizer on Dectape 172. The binary is called REC1 BINARY; the other files are also on that tape.

Paper tapes of the symbolics and binaries of both SHAPESHIFTER and the character recognizer are in the paper tape rack in Cruft 111. Listings will be next door. A description of each file and how to reload the files to create a new binary is in the file called DRCTRY DS.

2. Starting the program

Set the test word switches to 1 (see 13). sense switch 1 down (4), sense switch 4 down (11), and put all the switch box switches in the middle (off) position (10).

The program may be started from the monitor in the normal manner, by typing START LEWIS. Almost a minute is required to fetch it off the drum, and SHAPESHIFTER requires about another minute to perform its own internal initializations, so nothing will appear on the scopes for quite a long time. Normally, when the initializations are completed, console

lights come on indicating that sequence break channels 4, 16, and 17 are active, and images appear on all the scopes. If the sequence break system lights never come on, or they come on but nothing is being displayed on one or more of the scopes, something is awry. Check the switches in the back of the 340 cabinets, the mus-mul switch, and the dis-div switch. Note also the options in (11) and (13) below.

3. Restarting the program

In case of certain types of accidents and bugs in the program, SHAPESHIFTER may be restarted at any time at 20000_8. The lengthy initialization will be repeated.

4. The initial state

The program should now be ready to go. It has three principal states or phases of operation, which are usually entered in cyclic order: the ground state, in which the program begins; the entry state, in which the formulas are written on the RAND tablet; and the execution state, in which the range of the domain under the transformation is computed and displayed.

In the normal assignment of the scopes (but see 13) the scope over the RAND tablet will show the "control panel," the one in the center the domain in the complex plane, and the one to the right the range. The half-size of the initial window is 1, the center at Ø, as indicated by the co-ordinates displayed in the corners of the range scope.

If sense switch 1 was down during the initialization, the grid domain is seen; if sense switch 1 was up, the circle. This is the only point when the position of switch 1 is significant.

5. The Grafacon

The program should now be tracking the movement of the RAND tablet pen on the control scope. If it is not, make sure the RAND tablet has been turned on.

N.B. : in the following descriptions the words "press down the pen" should be interpreted as "press down the pen until the switch closes and then lift it up until the switch opens."

The microswitch in the tip of the pen sometimes breaks; keep this in mind if you experience unusual difficulty while using it.

6. Reading the trees

To read in the tree structure for the character recognizer, put the paper tape in the reader, turn on the reader, and then push the READ TREES button with the tablet pen. Tracking ceases and will resume when the trees have been read.

If the reader is off when the READ TREES button is pushed, the program will hang up. Return to step 3.

7. The "buttons"

The operation of the program is controlled by pushing the displayed messages with the pen, as though they were buttons. Some of these are self-explanatory, others are not.

RESTART GRID
RESTART CIRCLE recreate the initial domains.

ENTER causes the program to go into the character entry phase (8).

EDIT returns the program to the entry phase without erasing the existing formulas.

ORIGIN. The domain is displayed on the scope labelled "T" in the lower right-hand corner. The formulas written have Z as the independent variable. Z is related to T by the formula displayed on the control scope which initially reads " Z = T - (0.00,0.00) " . To change the bracketed number and thereby effect a pre-translation of the domain relative to the origin, do as follows:

a) Press the ORIGIN button. All tracking ceases.

b) Press the pen down anywhere. The cross on the domain scope will track the pen relative to the cross's original position, and the formula " Z = T - (... , ...) " will change to indicate the new value of the parameter.

c) Press down a third time to return control.

ABORT. During the execution phase, pressing down on this button causes the transformation and the partially-computed range to be scrubbed and control to return to the ground state.

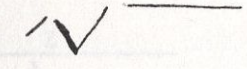
FIX. During the execution phase, pressing this button indicates that the result of the current transformation is to be saved and used as the next domain. After this button has been pressed, the transformation continues until the full range has been computed and then the program returns to the ground state.

Note: To abort after the FIX button has been pressed, press EDIT and then ABORT.

8. Character entry

In order for the character recognizer to function properly, sense switch 6 must be in the same position as it was in when the trees were created. The paper tape mentioned in (1) must be used with the switch up.

With these trees, the radical sign must be written in a somewhat peculiar manner, namely as four separate strokes:



That is , the pen must be lifted three times while you are writing this character.

With these trees, C is written as ¢ ; 7 as 7 ; and Z as Z. The scrub mark is written || .

To parse or abort, write a / over the appropriate button. This is actually my version of the "period" (.) in this set of trees.

If free storage is exceeded in the course of writing a formula, a message will be typed out and you will have to start over again.

The separator between formulas is the semicolon (written ;). It must be written to the right of a formula and the line which appears across the scope from the lowest point on the semicolon must indeed separate the formulas.

9. Parsing

If the parse is successful, the linear-string version of the formulas will appear at the bottom of the control scope and the transformation will proceed. If the characters cannot be parsed, the message "ILLEGAL FORMULA"will appear instead.

Common reasons for failure to parse are the following:

a) You have written something which is interpreted as an absurd exponentiation or subscripting, e.g. :

$$W = \frac{Z-1}{Z+1}$$

in which the expression is not on the same line with the "=" .

b) The characters which you intend to group with a division bar or radical sign are not so grouped, e.g. :

$$W = \cfrac{Z}{Z + \cfrac{1}{Z + \cfrac{1}{Z + 1}}}$$

(The last "1" is not under the next-to-last bar.)

c) The left side of the semicolon is to the left of the left edge of some other character in its formula:

$$A = Z + \frac{1}{Z} + 1 \;\Big/\; W = A + \frac{1}{A}$$

The parser sometimes fails from no apparent cause. Press ABORT and RESTART, and try again.

10. Scaling

During either the ground state or the execution state, the size and position of the window through which the domain is viewed may by changed. To change the size, hold closed switch 0 or 1 of the switch box (0 increases the size, 1 decreases it). The co-ordinates of the corners of the window are displayed. To move the center of the window, use the joystick (11).

The following situation occurs frequently enough to merit description. The program is halfway through a lengthy computation when you decide that there is detail outside of the window.which you would like to see. It is much faster to press FIX and wait for the "fix" to occur, and then do the scaling, than to do it as the points are being computed.

11. Joystick option

If you wish to use the joystick (the moving one) for changing the center of the viewing window, set sense switch 4, connect the joystick to channels 0 and 1 of the A/D converter, and turn on the A/D converter before starting the program. The window will move in the direction in which the shaft is pushed until the joystick is released and allowed to return to its rest position.

12. Crashing (i.e. returning to the monitor)

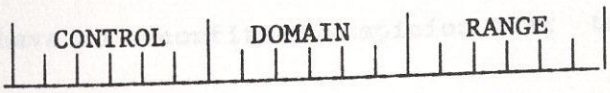
It is generally impossible to crash from the console except by first entering the character entry phase or by using the UTR .

13. Reassigning the scopes

If 1 is in the test word when the program is started, the assignment of the scopes at initialization is as described.

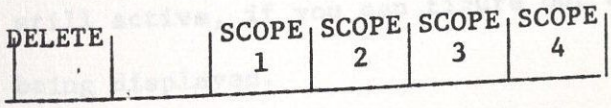
At any time except during the character entry phase, the assignment of pictures to scopes and the choice of pictures to be displayed may be changed. By displaying only one of the three pictures, the annoying flicker can be eliminated.

Consider the test word to be three groups of six switches:



The left-hand group of switches applies to the "control panel" picture, the center to the domain picture, the right to the range picture.

For each group, the switches have the following effects:

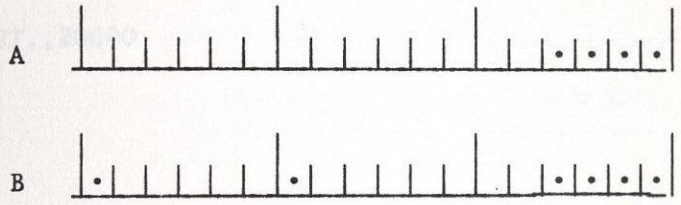


If the left-hand switch is raised, the picture is removed from the display

file. If the left-hand switch is down, the four right-hand switches determine the scopes on which the picture will appear: up means display on that scope, down means do not.

While changing the switches, do so in such a way that something is always being displayed. Failure to do so sometimes causes a hang-up.

Note well the difference between the following switch settings:



In each case, the range will be displayed on all four scopes. But in case A, the display will flicker badly; in case B, not at all.

14. Bugs

In addition to the bugs already noted, there are a few others of which I have become aware.

a) Some of the computational subroutines have bugs. In particular, $\sinh^{-1}$ works correctly in most cases, but sometimes causes the computer to halt. Press continue. I have an unconfirmed suspicion that tan does not work properly at all.

b) If the linear-string version of the formulas is more than one line long, the display will disappear. Restart.

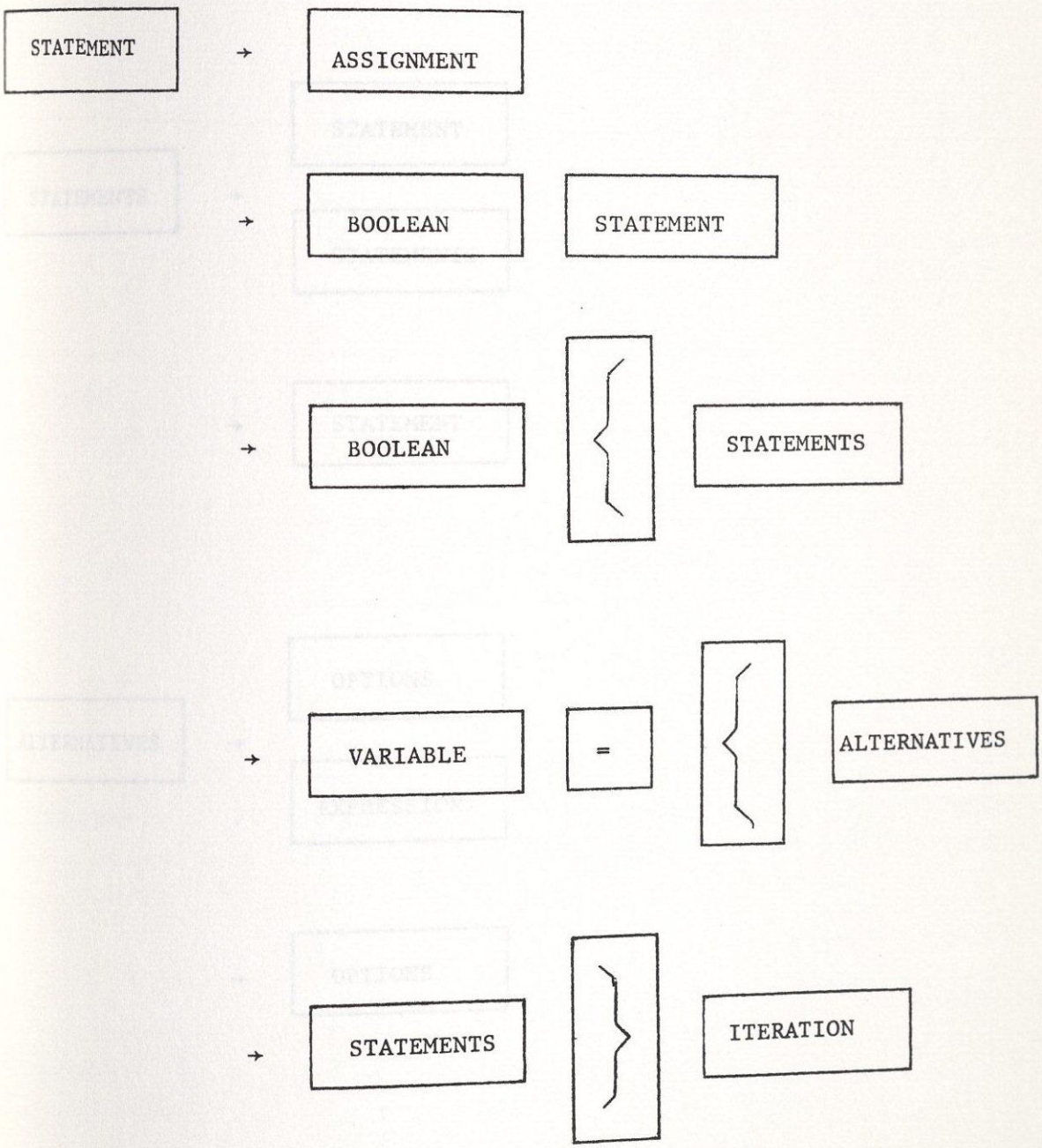
c) After editing and reparsing, the RESTART and ORIGIN buttons do not reappear. They are still active, if you can figure out where they are; they are simply not being displayed.

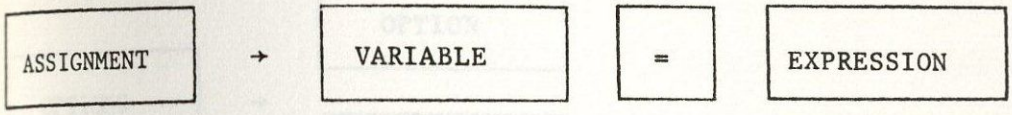
15. Patching and saving.

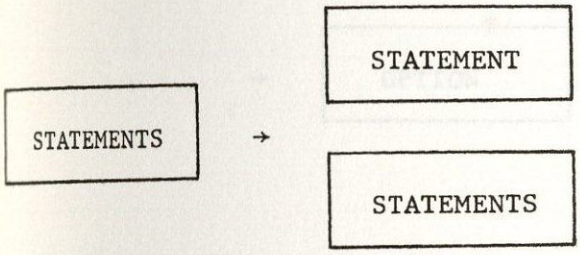
SHAPESHIFTER can be examined with DDT. Unfortunately, an attempt to save a patch by typing SAVE LEWIS, ... causes the monitor to hang up and the binary on the drum to be destroyed. (This is a system bug.) Instead, type SAVE PATCH, ... and restart by typing

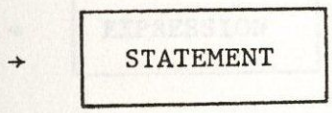
```
FETCH LEWIS
FETCH PATCH
START,,20000
```

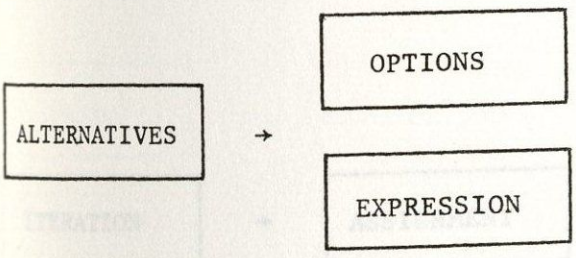
Appendix 8. Syntax for Two-Dimensional Programming Language

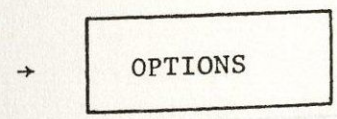


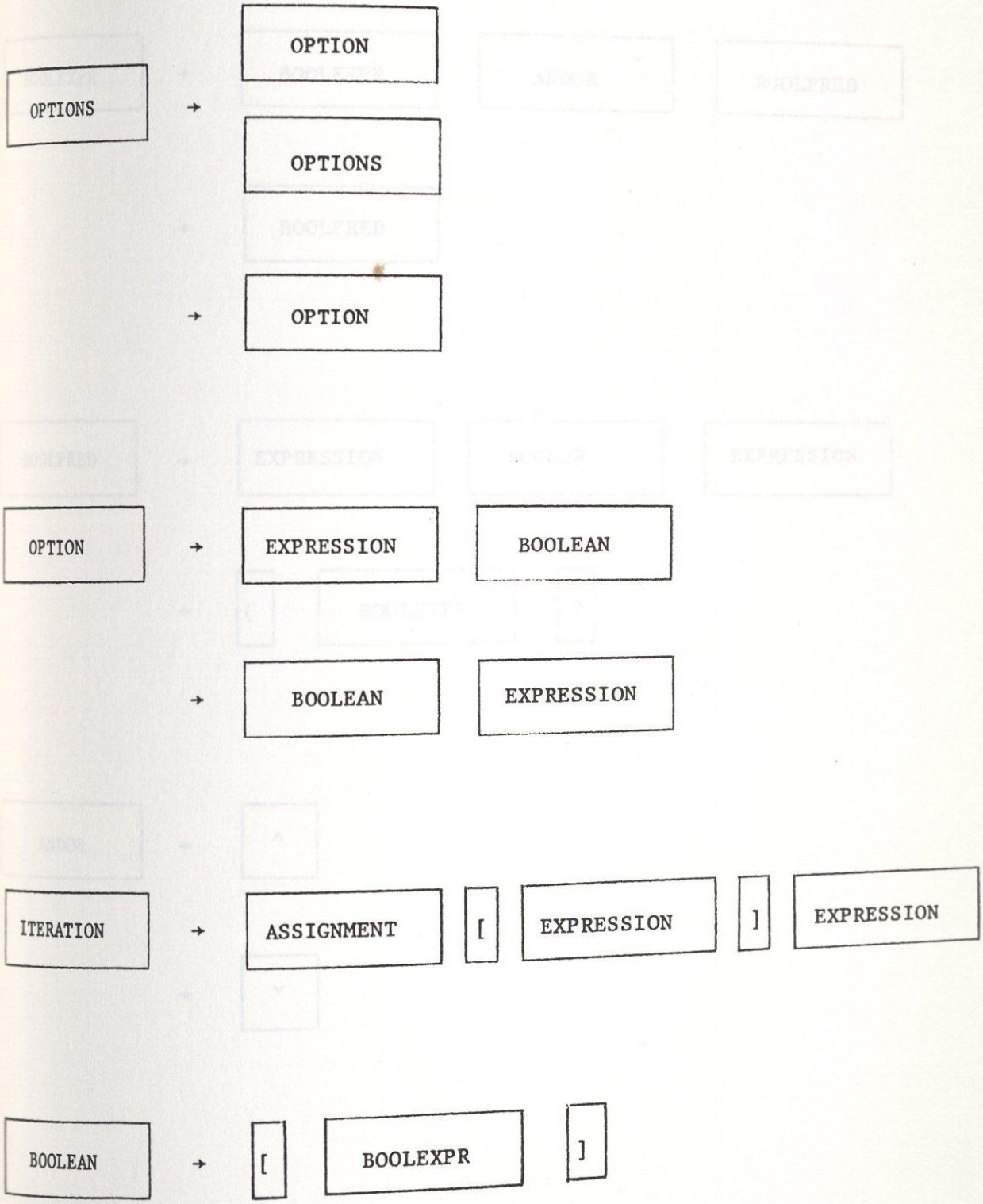

ASSIGNMENT
→
VARIABLE
=
EXPRESSION

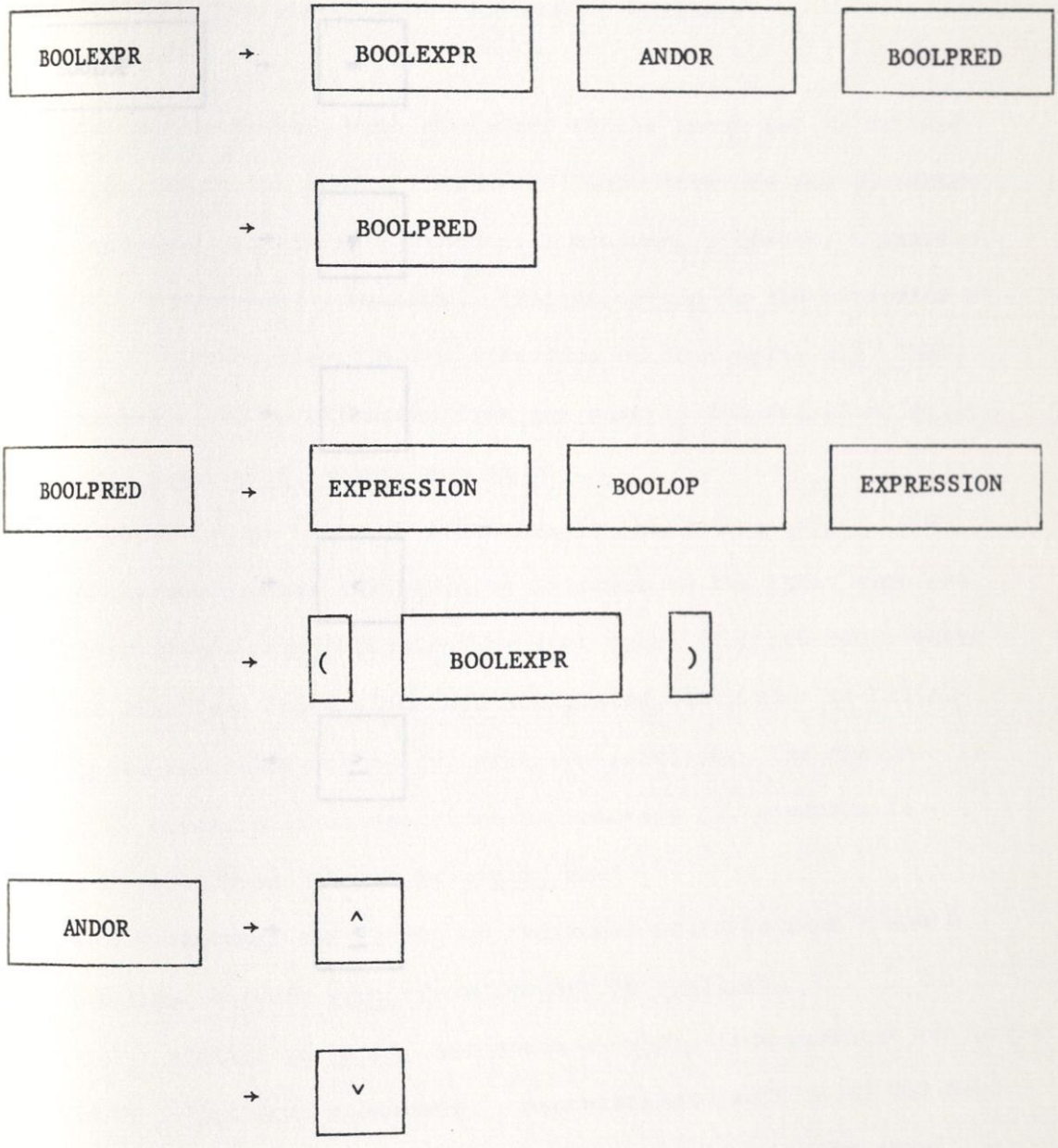

STATEMENTS
→
STATEMENT
STATEMENTS

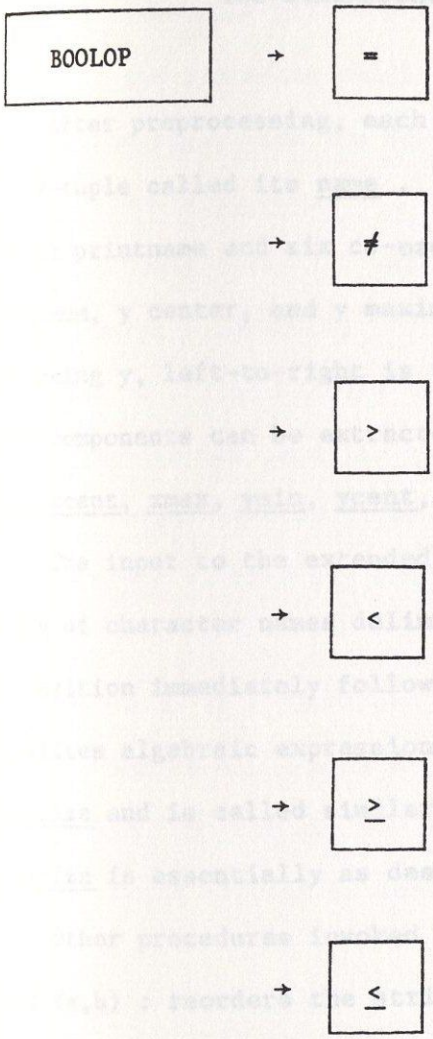

→
STATEMENT


ALTERNATIVES
→
OPTIONS
EXPRESSION


→
OPTIONS


OPTIONS
OPTION
OPTIONS
OPTION
OPTION
EXPRESSION
BOOLEAN
BOOLEAN
EXPRESSION
ITERATION
ASSIGNMENT
[
EXPRESSION
]
EXPRESSION
BOOLEAN
[
BOOLEXPR
]


BOOLEXPR
→
BOOLEXPR
ANDOR
BOOLPRED
→
BOOLPRED
BOOLPRED
→
EXPRESSION
BOOLOP
EXPRESSION
→
(
BOOLEXPR
)
ANDOR
→
^
→
v


BOOLOP
=
≠
>
<
≥
≤

Appendix 9. Recognition Algorithm for Two-Dimensional Programming Language

After preprocessing, each character in the input set is defined by a 7-tuple called its name . The seven components are the character code or printname and six co-ordinates: x minimum, x center, x maximum, y minimum, y center, and y maximum. (Bottom-to-top is the direction of increasing y, left-to-right is the direction of increasing x.) These seven components can be extracted from the name by the functions print, xmin, xcent, xmax, ymin, ycent, and ymax.

The input to the extended linearization procedure expand is a string of character names delimited by pointers to the first name and the position immediately following the last name. The procedure which linearizes algebraic expressions and assignment statements is called linearize and is called similarly, with two pointers. The routine linearize is essentially as described by Anderson [2, Appendix 1].

Other procedures invoked by expand are:

ysort (a,b) : reorders the string of character names between a and b according to their ymax, from largest to smallest.

xsort (a,b) : similar to ysort, but sorts by xmin, from smallest to largest.

gnexst (a,b) : "get next statement." Assumes that ysort (a,b) has been performed. Scans along the input string (i.e. down the page) until a sufficiently wide space occurs on the page with no character in it. Returns with the pointer to the string position following the name of the last character in the group. Also checks whether any one character has both the largest ymax and the smallest ymin in the group; if there is any such character, its name is called the

binder.

name(a) is the name of the character pointed to by the pointer a.

output is the procedure which places a character or string of characters on the right-hand end of the output string. Identical to Anderson's append.

findfor/findback (a,b,c) : searches the string between a and b forwards / backwards (left-to-right / right-to-left) for the name c . The failure to find the name c is an error condition, except in one case (so noted on the flowchart).

pinv(a) : "print-inverse." Is used in conjunction with findfor and findback to find any character name c such that print (c) = a.

Figure 20 is a flowchart for expand. Note that throughout the procedure and at all depths of recursion only one string of character names exists. By sorting and searching within this string, a minimal amount of extra storage is used.

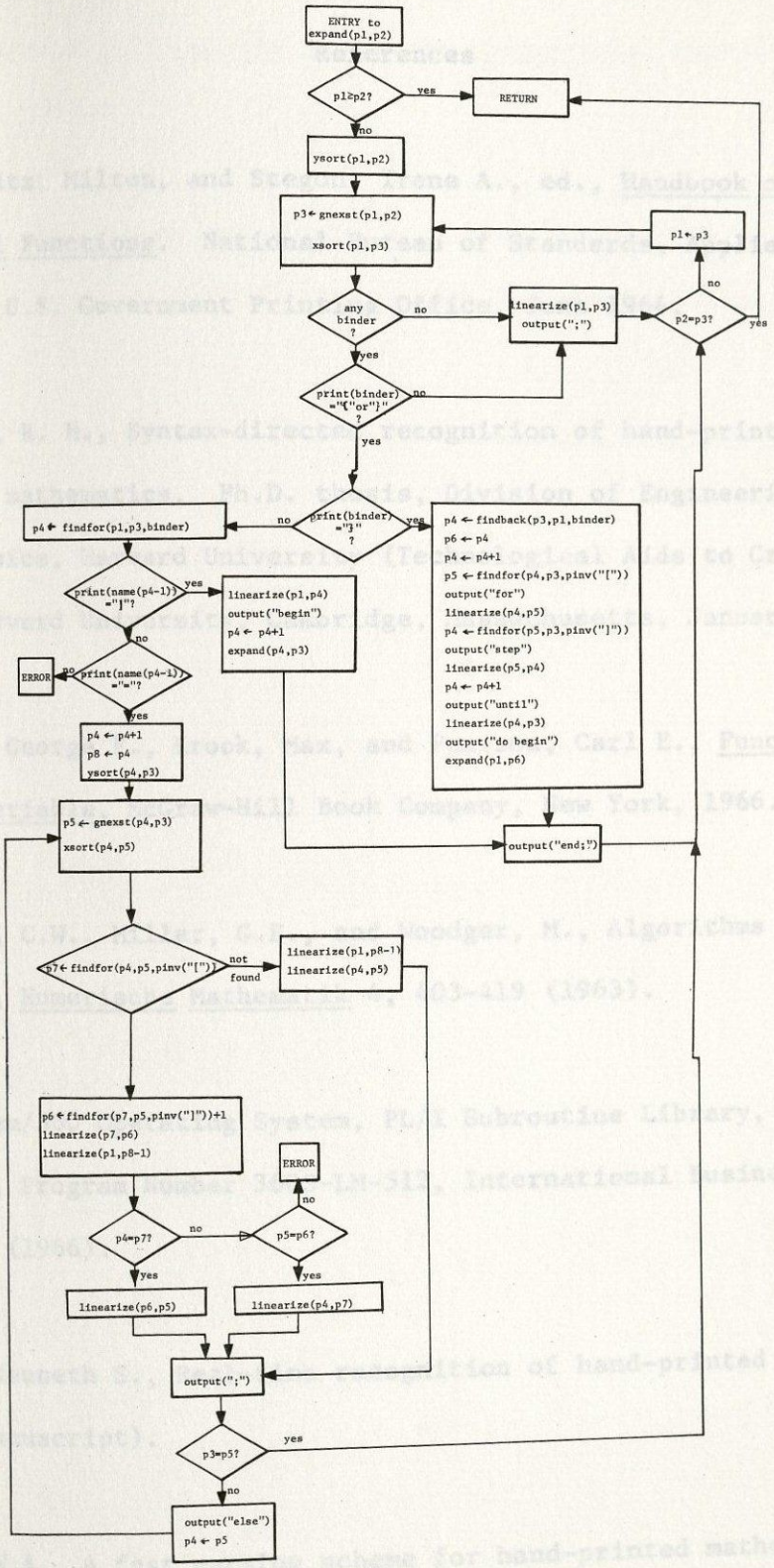

ENTRY to expand(p1,p2)
p1≥p2?
yes
RETURN
no
ysort(p1,p2)
p3 ← gnexst(p1,p2)
xsort(p1,p3)
p1 ← p3
any binder ?
no
linearize(p1,p3)
output(";")
p2=p3?
no
yes
yes
print(binder) ="{"or"}" ?
no
yes
p4 ← findfor(p1,p3,binder)
no
print(binder) ="3" ?
yes
p4 ← findback(p3,p1,binder)
p6 ← p4
p4 ← p4+1
p5 ← findfor(p4,p3,pinv("["))
output("for")
linearize(p4,p5)
p4 ← findfor(p5,p3,pinv("]"))
output("step")
linearize(p5,p4)
p4 ← p4+1
output("until")
linearize(p4,p3)
output("do begin")
expand(p1,p6)
print(name(p4-1)) ="]"?
yes
linearize(p1,p4)
output("begin")
p4 ← p4+1
expand(p4,p3)
no
ERROR
no
print(name(p4-1)) =")"?
yes
p4 ← p4+1
p8 ← p4
ysort(p4,p3)
p5 ← gnexst(p4,p3)
xsort(p4,p5)
output("end;")
p7 ← findfor(p4,p5,pinv("["))
not found
linearize(p1,p8-1)
linearize(p4,p5)
p6 ← findfor(p7,p5,pinv("]"))+1
linearize(p7,p6)
linearize(p1,p8-1)
ERROR
p4=p7?
no
p5=p6?
no
yes
yes
linearize(p6,p5)
linearize(p4,p7)
output(";")
p3=p5?
yes
no
output("else")
p4 ← p5

Figure 20

References

1. Ambramowitz, Milton, and Stegun, Irene A., ed., Handbook of Mathematical Functions. National Bureau of Standards, Applied Mathematics Series, 55, U.S. Government Printing Office, June 1964.

2. Anderson, R. H., Syntax-directed recognition of hand-printed two-dimensional mathematics. Ph.D. thesis, Division of Engineering and Applied Physics, Harvard University (Technological Aids to Creative Thought, Harvard University, Cambridge, Massachusetts, January 1968).

3. Carrier, George F., Krook, Max, and Pearson, Carl E., Functions of a Complex Variable, McGraw-Hill Book Company, New York, 1966.

4. Clenshaw, C.W., Miller, G.F., and Woodger, M., Algorithms for special functions I, Numerische Mathematik 4, 403-419 (1963).

5. IBM System/360 Operating System, PL/I Subroutine Library, Computational Subroutines, Program Number 360S-LM-512, International Business Machines Corporation (1966).

6. Ledeen, Kenneth S., Real-time recognition of hand-printed input (informal manuscript).

7. Martin, W.A., A fast parsing scheme for hand-printed mathematical expressions. MAC-M-360, Project MAC, Massachusetts Institute of Technology (October 19, 1967).

8. Nixon, Floyd E., Principles of Automatic Controls. MacMillan and Co., London, 1958.

9. Wirth, N., and Weber, H., EULER: a generalization of Algol, and its formal definition: part I. Comm. ACM, 9 (January 1966), 13-25.